System design: Google maps

Suresh Podeti
6 min read · Oct 6, 2023

Photo by Thomas Kinto on Unsplash

Requirements

Functional

- **Identify the location:** Users should be able to approximate their location (latitude and longitude in decimal values) on the world map
- **Recommend the fastest route:** Given the source and destination (place names in text), the system should recommend the optimal route by distance and time, depending on the type of transportation

Non-Functional

- **Availability:** The system should be highly available.
- **Scalability:** It should be scalable because both individuals and other enterprise applications like Uber and Lyft use Google Maps to find appropriate routes.
- **Accuracy:** The ETA we predict should not deviate too much from the actual travel time.
- **Less response time:** It shouldn't take more than two or three seconds to calculate the ETA and the route, given the source and the destination points

Design

Workflow

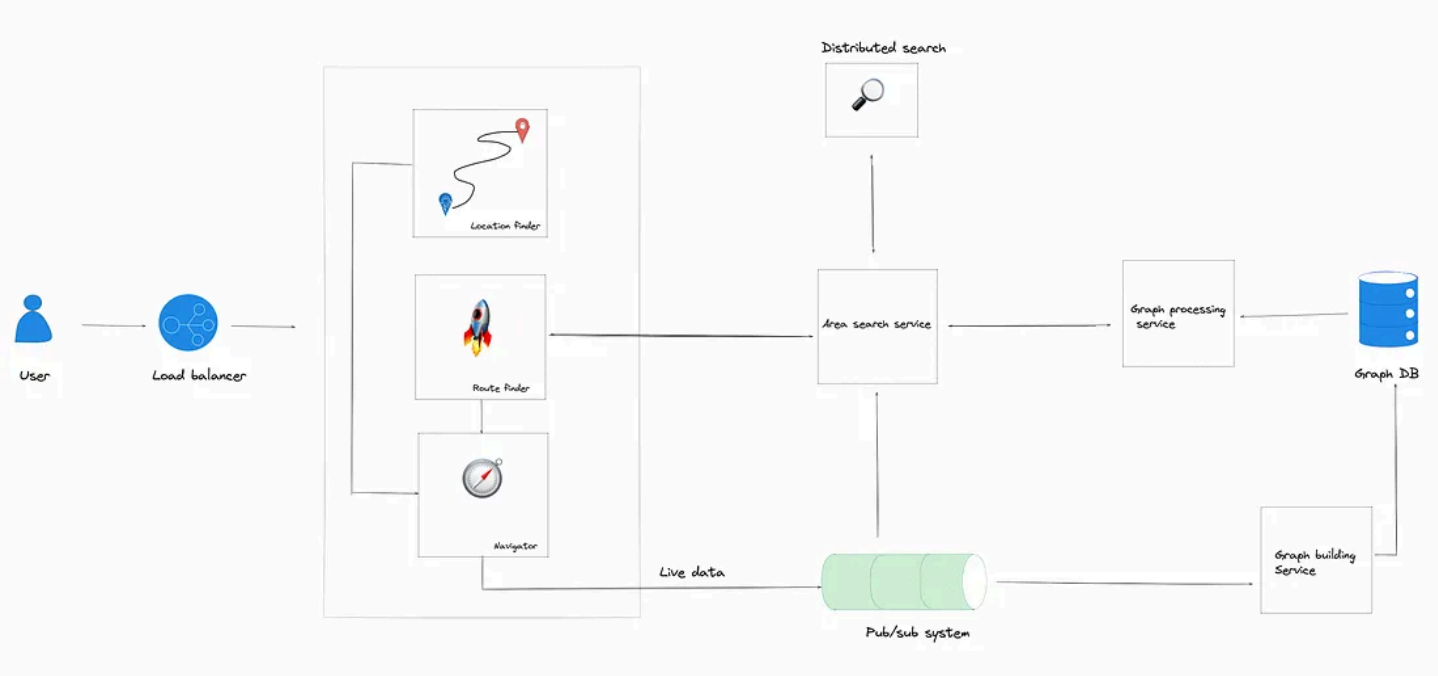
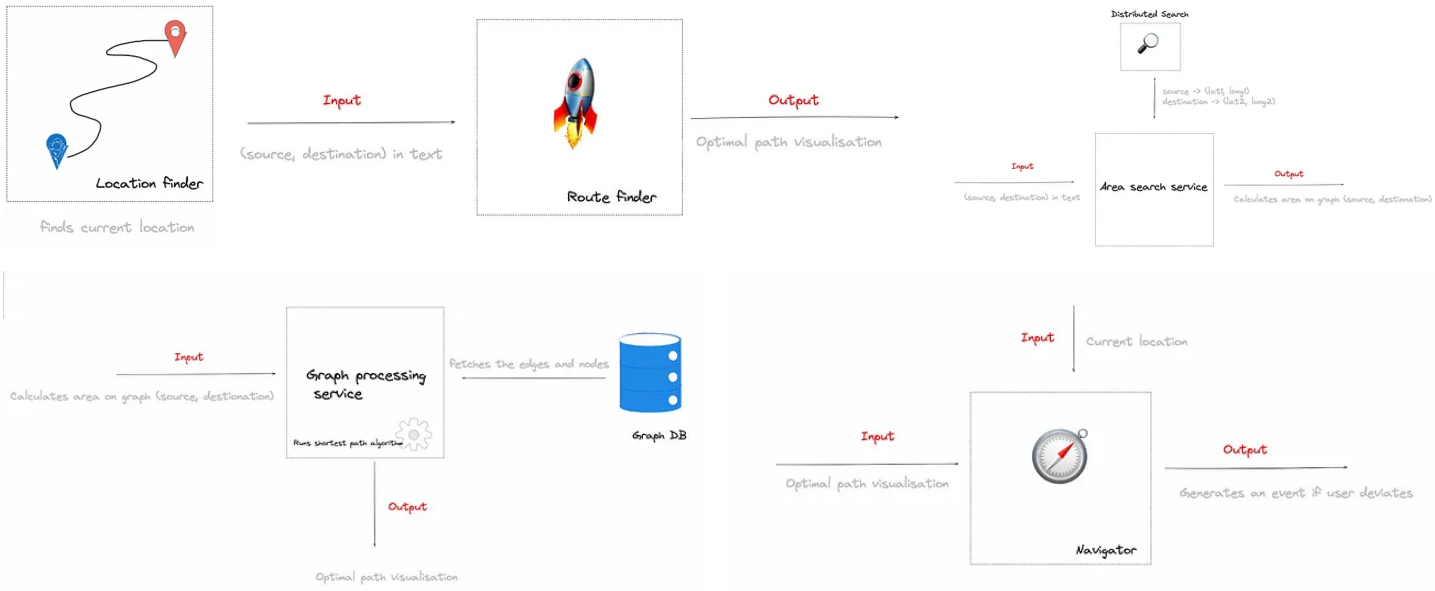


Fig 1.0: A high-level design of a map system

- **Location finder** determines the user's current location using GPS, Wi-Fi, and cellular technology, and show it on the map since we can't possibly remember latitude and longitude for every place in the world.
- User types destination point as an address in text format, and request is forwarded to the **route-finder service**.
- The route finder forwards the request to an **area search service** with source and the destination points.

- The area search service uses the **distributed search service** to find the latitude/longitude for the source and destination. It then calculates the area on the map spanning the two (source's and destination's) latitude/longitude points.
- After finding the area, the area search service asks the **graph processing service** to process part of the graph.
- The graph processing service fetches the edges and nodes within that specified area from the **graph database,** finds the shortest path, and returns it to the route-finder service that visualises the optimal path. It also displays the steps that user should follow for navigation.
- Now that the user can visualize the shortest path on the map, they also want to get directions towards the destination. The direction request is handled by the **navigator**. The navigator tracks the user path, and updates the user location on the map while the user is moving.
- If a user deviates from the path, it generates an event that is fed to **Pub-sub system.**
- Upon receiving the event from the navigator, area search service recalculates the optimal path and suggest it to the user. The navigator also provides a stream of live location data to the graph, that can be used to improve route suggestions provided to the users

Components



Challenges

Scalability

Graph with billions of vertices and edges and we have to traverse the whole graph to find the shortest path, and the key challenges are inefficient loading, updating, and performing computations.

Solution — Break down a large graph into smaller subgraphs, or partitions are called **segments**, and each segment corresponds to a subgraph. Each intersection/junction acts as a vertex and each road acts as an edge. The graph is weighted, and there could be multiple weights on each edge — such as distance, time, and traffic — to find the optimal path.

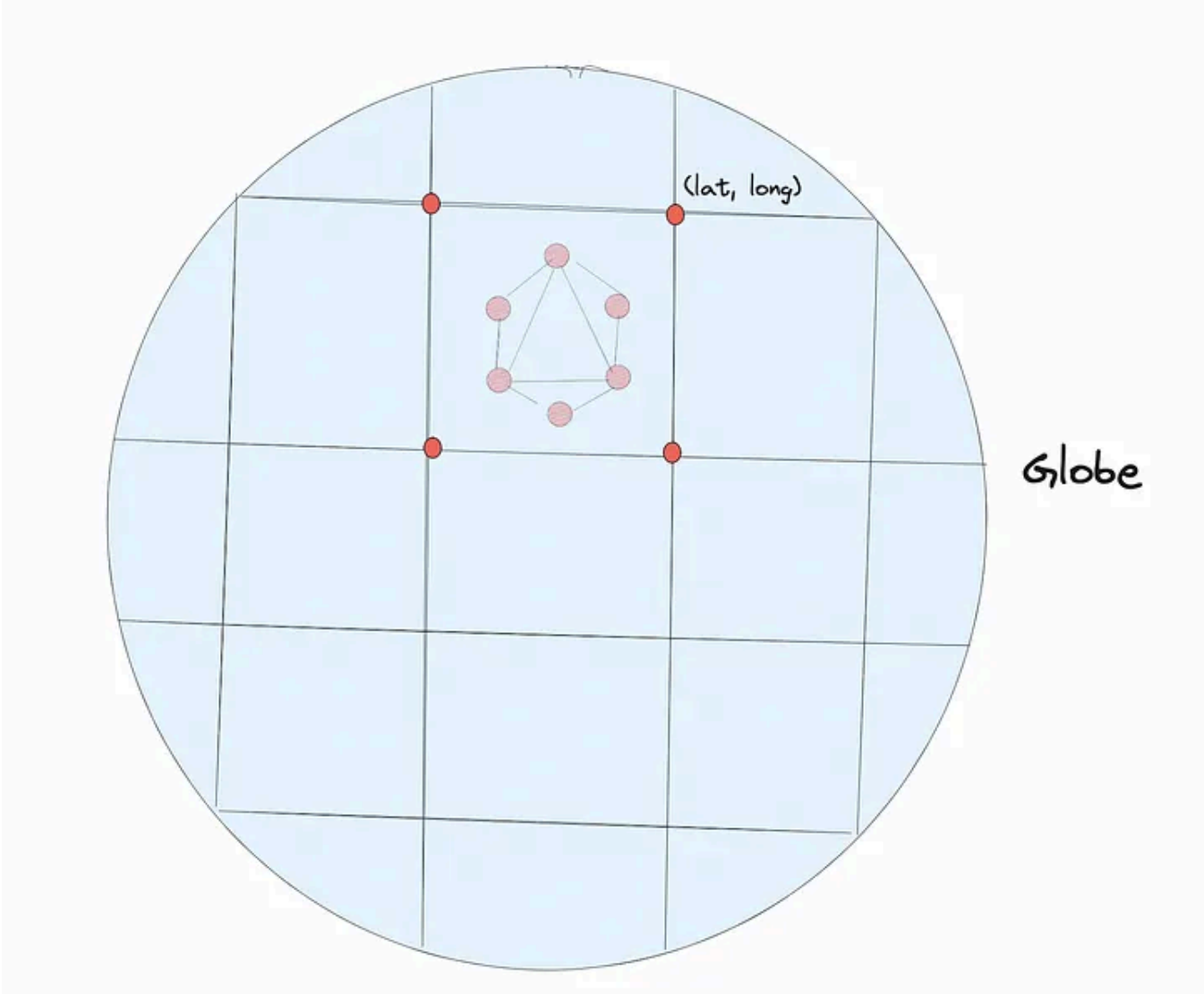


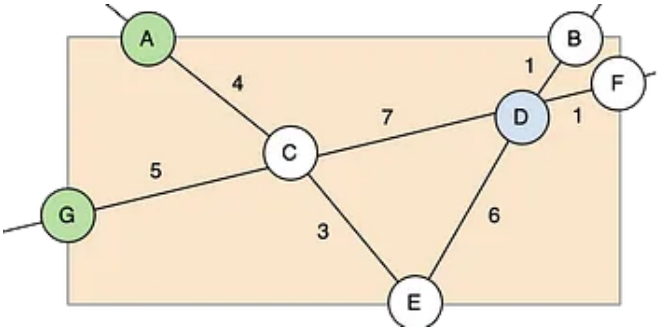
Fig 2.0: Entire globe is split into segments

Locations within a segment

Finding paths within these segments works because the segment's road network graph is small and can be loaded easily in the main memory, updated, and

traversed. The most common shortest path algorithm used to find shortest path is the **Dijkstra's algorithm**.

After running the shortest path algorithm on the segment's graph, we store the algorithm's output in a distributed storage to avoid recalculation and cache the most requested routes. The algorithm's output is the shortest distance in meters or miles between every two vertices in the graph, the time it takes to travel via the shortest path, and the list of vertices along every shortest path. All of the above processing (running the shortest path algorithm on the segment graph) is done offline (not on a user's critical path)



Cached Information: output of the shortest path algorithm

Source	Destination	Distance (m)	ETA (min)	Path
A	D	11	15	A->C->D
G	D	12	16.5	G->C->
⋮	⋮	⋮	⋮	⋮

Fig 3.0: Finding a path between vertices within a segment

Locations lie on the edges

If two points lies on the edges of the **graph** (Note: Not on segment edges), find the vertices of the edge on which the points lie, calculate the distance of the point from the identified vertices, and choose the vertices that make the shorter total distance between the source and the destination.

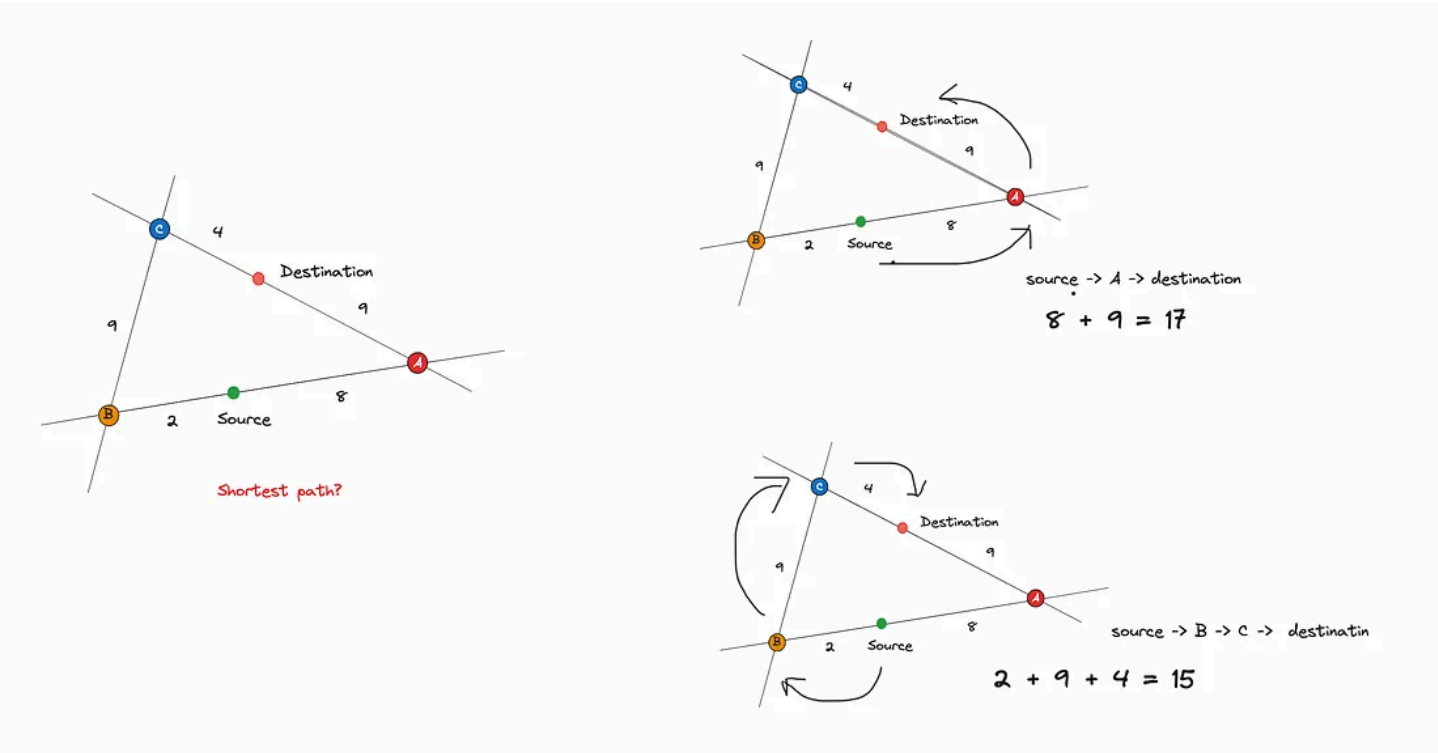


Fig 4.0: Finding shortest path of locations lie on edges

Locations belong to two different segments

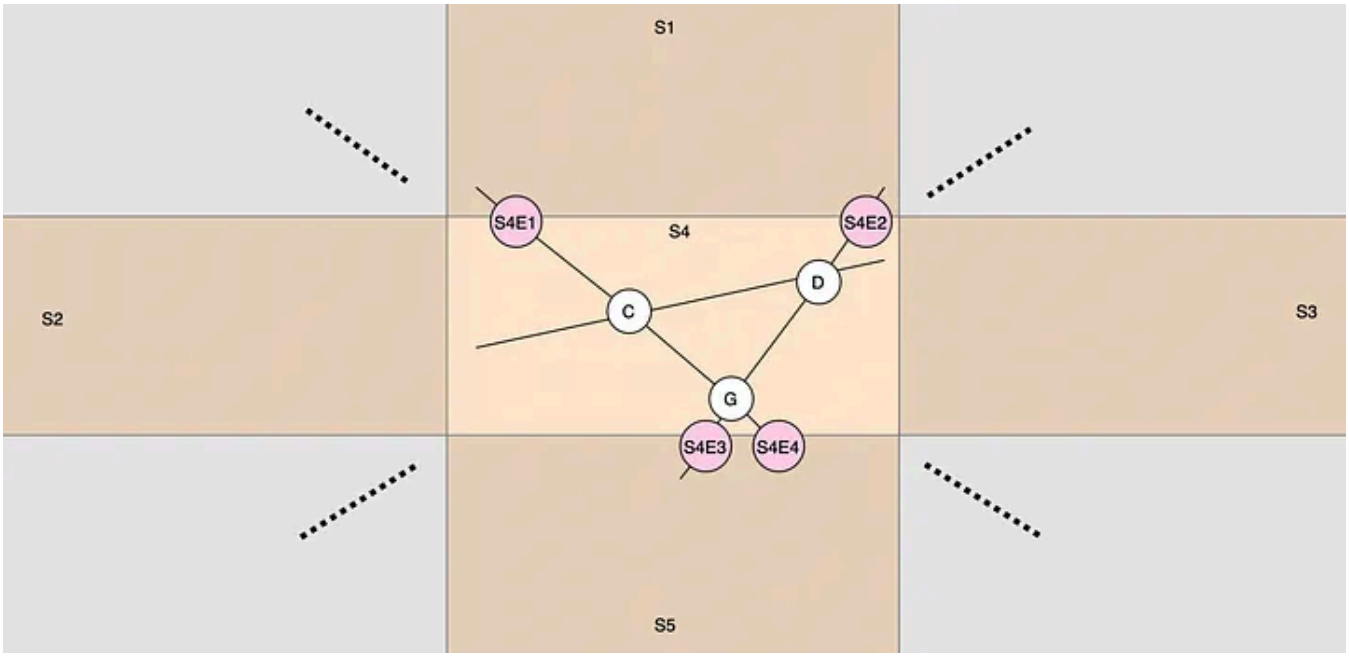


Fig 5.0: Connecting the segments by considering exit points of a segment as vertices

Given the source and the destination, we can find the segments in which they lie. For each segment, there are some boundary edges, which we call **exit points,** it connects neighboring segments and is shared between them.

For each segment, we calculate the shortest path between exit points in addition to the shortest path from the exit points to vertices inside the segment. Each vertex's shortest path information from the segment's exit points is cached.

We don't care about the inside segment graph. We just need exit points and the cached information about the exit points. We can visualize it as a graph made up of exiting vertices.

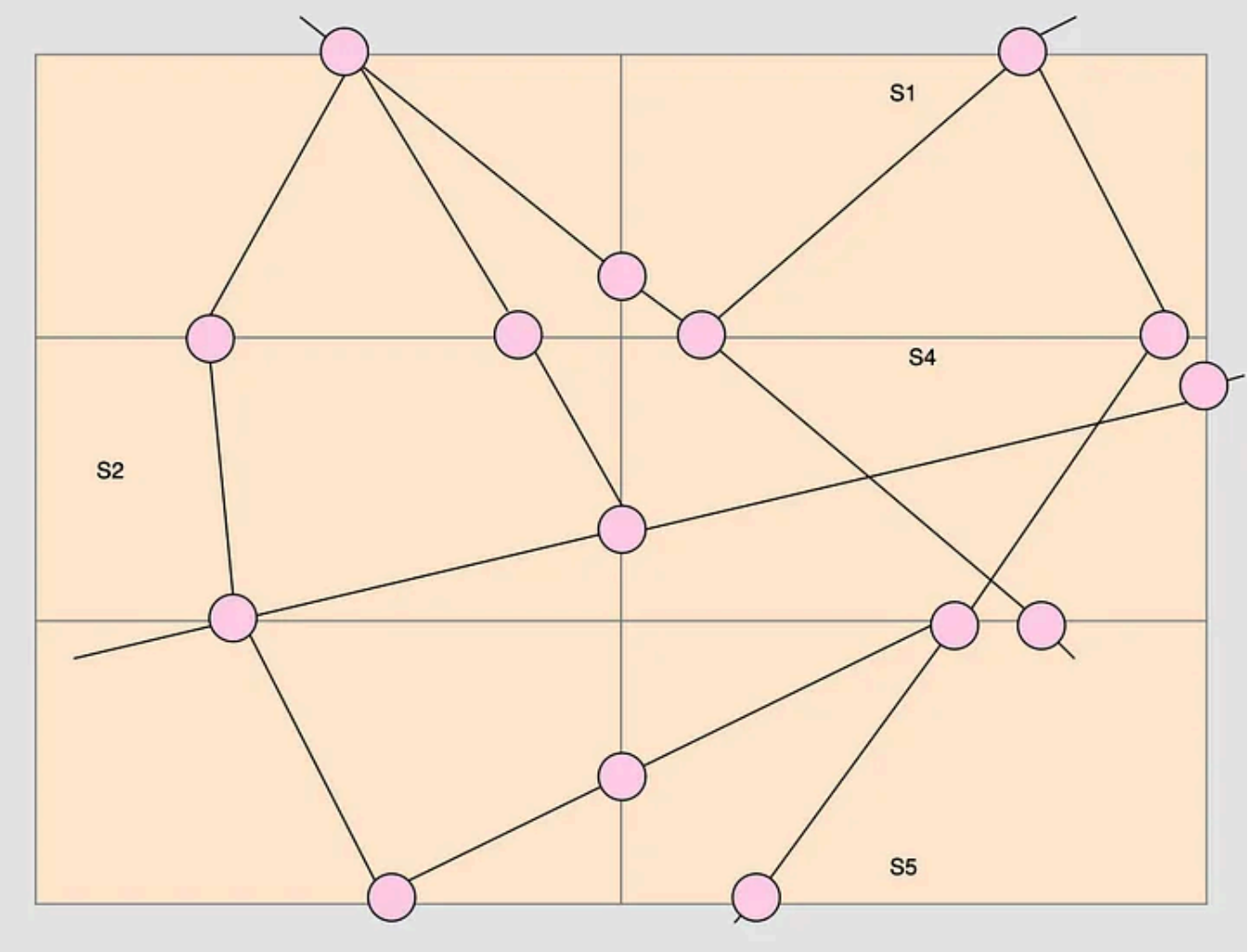


Fig 6.0: A graph made up of exit points, where the lines connecting the exit points are not actually straight. There could be many vertices in between the two exit points

It's critical to figure out how many segments we need to consider for our algorithm while traveling inter-segment. The aerial distance between the two places is used to limit the number of segments. We can include segments that are at certain arial distance from the source and destination in each direction. This is a significant improvement over the large graph.

Detailed Design

Storage schema

We need following database:

- **key-value store** — to store segment related information like segment boundaries, neighbours, and server details in which segment lies.
- **Graph database** — to store the graph corresponding to a segment

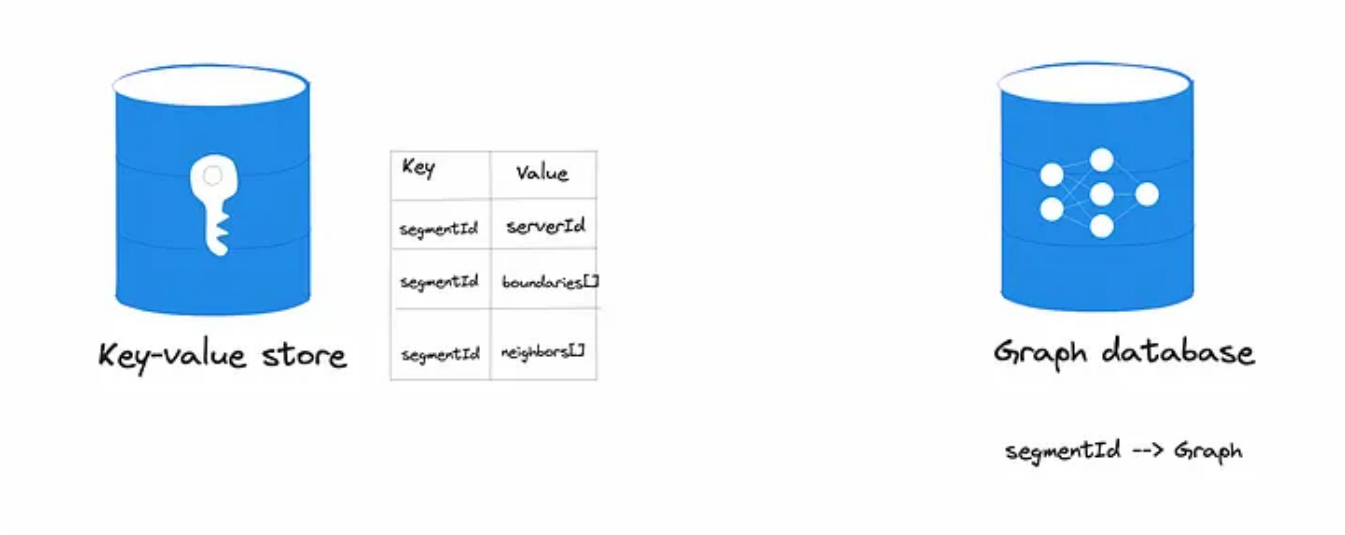


Fig 7.0: Databases

Design

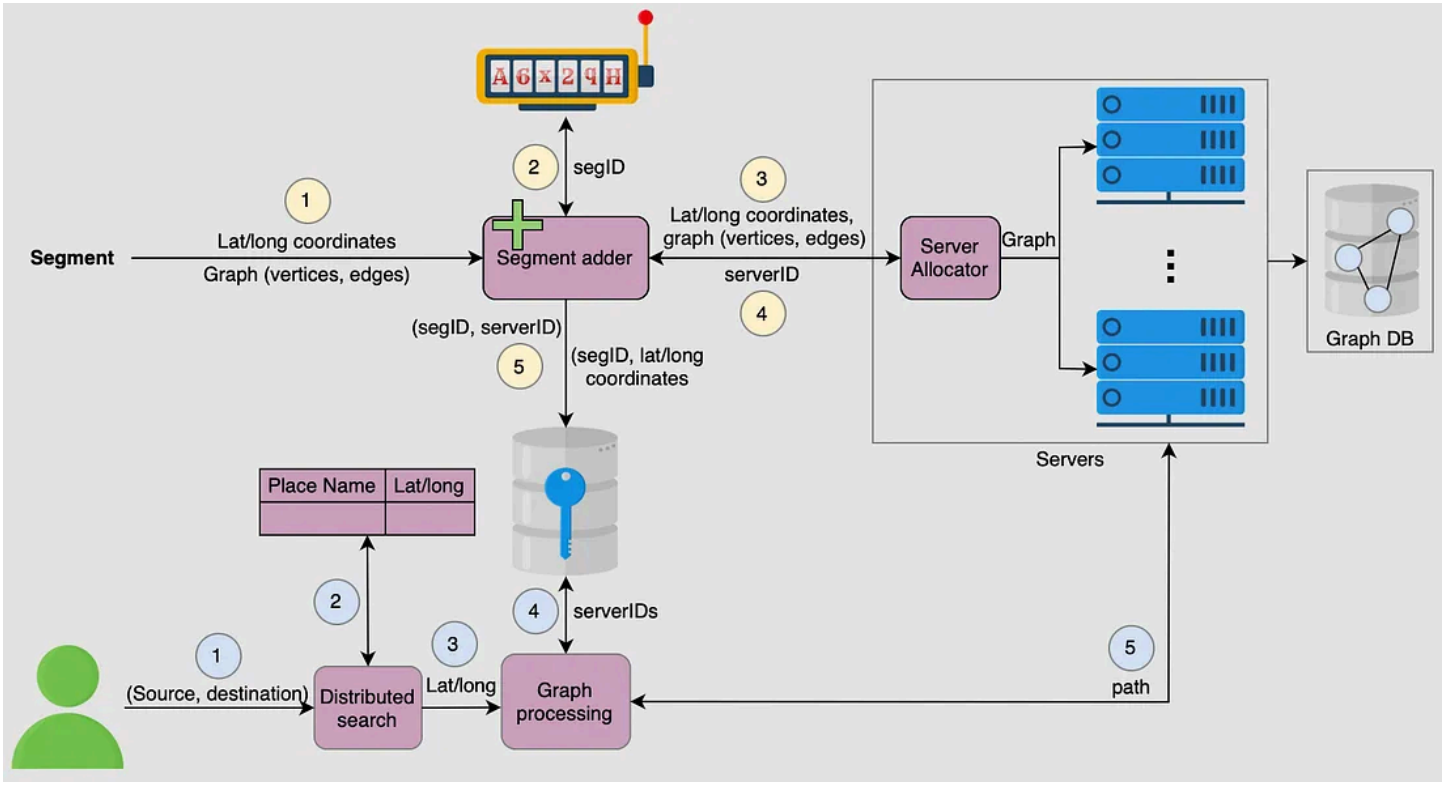


Fig 8.0: Workflow 1) Adding segments (in yellow), and workflow 2) Processing user requests (in blue)

Evaluation

Scalability

We divided the world into small segments. Each segment is hosted on a different server in the distributed system. The user requests for different routes are served from the different segment servers. In this way, we can serve millions of user requests.

It wouldn't have been possible to serve millions of user requests if we had a single large graph spanning the whole road network. There would have been memory issues loading and processing a huge graph.

We can also add more segments easily because we don't have to change the complete graph. We can further improve scalability by non-uniformly selecting the size of a segment — selecting smaller segment sizes in densely connected areas and bigger segments for the outskirts.

Smaller response times

We're running the user requests on small subgraphs. Processing a small subgraph of hundreds of vertices is far faster than a graph of millions of vertices. We can cache the processed small subgraph in the main memory and quickly respond to user requests. This is how our system responds to the user in less time.

There's another aspect that helps our system to respond quickly, and that is keeping the segment information in the key-value store. The key-value store helps different services to get the required information quickly.

Google Maps | Software Architect | Software Architecture | Distributed System Design | System Design Interview

Edit profile